Layout-Based Substitution Tree Indexing and Retrieval for Mathematical Expressions

Thomas Schellenberg[a], Bo Yuan[b] and Richard Zanibbi[c]

[a,c]Dept. of Computer Science
[b]Dept. of Networking, Security, and Systems Administration
Rochester Institute of Technology, 1 Lomb Memorial Drive, Rochester, NY, USA

ABSTRACT

We introduce a new system for layout-based (LaTeX) indexing and retrieving mathematical expressions using substitution trees. Substitution trees can efficiently store and find expressions based on the similarity of their symbols, symbol layout, sub-expressions and size. We describe our novel design and some of our contributions to the substitution tree indexing and retrieval algorithms. We provide an experiment testing our system against the TF-IDF keyword-based system of Zanibbi and Yuan and demonstrate that, in many cases, the quality of search results returned by both systems are comparable (overall means, substitution tree vs. keyword-based: 100% vs. 89% for top 1; 48% vs. 51% for top 5; 22% vs. 28% for top 20). Overall, we present a promising first attempt at layout-based substitution tree indexing and retrieval for mathematical expressions and believe that this method will prove beneficial to the field of mathematical information retrieval as a whole.

Keywords: Symbol layout tree, substitution tree, information retrieval, mathematical expression, LaTeX

1. INTRODUCTION

While information retrieval is a well-explored field of research, mathematical information retrieval (MIR) is far less developed. We present a new MIR system that uses substitution trees[1] to index mathematical expressions from a database of LaTeX documents and retrieve relevant search results. The substitution tree data structure, originally created for automated theorem proving, has never before been used for MIR with layout-based expressions such as those written in LaTeX. This novel approach to MIR is comparable to existing MIR systems and promises to be even more effective upon further refinement.

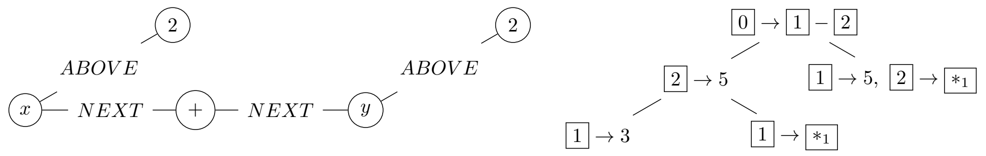
Conventional search engines remain ineffective in finding mathematical expressions due, in part, to their inability to correctly handle special mathematical notation and symbols.[2] Some have tried building search engines focusing on MIR to specifically address these issues, with varying success.[3] Notably, Kohlhase and Sucan made a search engine using substitution trees as well,[4] but for Content MathML (which represents mathematical semantics – see operator trees in Zanibbi and Blostein[3]). We chose to index LaTeX because it is a more popular and available source for mathematical information, though our system could be modified to index other encodings such as Presentation MathML or images. Zanibbi and Yuan have created a keyword-based MIR system using Lucene that they show to be effective.[5] It uses a vector-space approach to store LaTeX expressions and a term frequency-inverse document frequency (TF-IDF) model to rank search results. We will be using their system in our tests for comparison.

2. SUBSTITUTION TREES

Graf first introduced the substitution tree as a tool for automated theorem proving.[1] It provides an efficient and intuitive way for storing predicates and grouping them based on similarity. These predicates are composed of terms, and the similarity of two predicates is based upon the terms they share and the arrangement of those terms. Our implementation uses mathematical expressions instead of predicates; expressions themselves are essentially sets of terms, where each term represents a symbol in the expression. This makes our conversion much easier:

Further author information: [a]mtschellenberg@gmail.com, [b]bxyics@rit.edu, [c]rlaz@cs.rit.edu (Send mail to [a]).
This material is based upon work supported by the National Science Foundation under Grant No. IIS-1016815.

Figure 1. *Left*: The symbol layout tree for $x^2 + y^2$. The "above" branch is used for terms that are superscript to another term (a caret in LaTeX), the "below" branch for terms that are subscript (an underscore), and the "argument" branches for terms that are contained within another term (LaTeX functions like `\sin` (1 branch) or `\frac` (2 branches)).
Right: A substitution tree representing the expressions $x - 5$, $3 - 5$ and $5 - x$. These expressions are normalized to $\boxed{*_1} - 5$, $3 - 5$, and $5 - \boxed{*_1}$ respectively. Indicator variables can be substituted for any single mathematical variable; in this case, $\boxed{*_1} \to x$. $\boxed{0}$ is used as the root substitution variable. An expression can be reconstructed by composing the substitutions along a branch. For example, $\boxed{0} \to \boxed{1} - \boxed{2}$, $\boxed{2} \to 5$, $\boxed{1} \to \boxed{*_1}$ where $\boxed{*_1} \to x$ produces $x - 5$.



each node in our substitution trees represents an expression instead of a predicate, but otherwise their behavior is much the same. Therefore we are specializing – rather than generalizing – Graf's implementation.

In a substitution tree, each node represents an expression; the leaf nodes represent specific expressions that have been inserted into the tree, while the non-leaf nodes represent expressions containing one or more generalized variables, known as *substitution variables*. Each child node specializes its parent by replacing one or more of these substitution variables with specific terms; these replacements are called *substitutions* and are contained within the node. Thus the expressions in the tree go from more abstract to more specific through substitutions. Following a branch of the tree from root to leaf will produce an expression represented by the leaf node through the combined substitutions of every node along the branch. This also means that every child node is an *instance* of its parent, requiring only a substitution (or set of substitutions) to produce the child's expression from the parent's, and every parent node is a *generalization* of each of its children. The tree represents all of the inserted expressions simultaneously and each expression can be produced by following a specific branch.

We use another type of tree, called a *symbol layout tree* (or SLT), to represent the expressions within the nodes of the substitution tree. An SLT is an encoding for a mathematical expression that is constructed based on the spatial relationship of the symbols in that expression. Each node of the SLT contains a *term* that represents a variable, constant, symbol or function in an expression. Each node has at least four branches dependent on the node's term: one representing the term(s) positioned spatially to the right of the term, one representing the term(s) positioned spatially above or superscript to the term, one representing the term(s) positioned spatially below or subscript to the term, and one or more representing the sets of terms that are argument(s) to the term, if the term is a function: in LaTeX functions (e.g., `\frac`), these are positioned within sets of brackets { }.

2.1 Definitions

A *symbol layout tree* is described by a 5-tuple $(t, A, B, \{X_1, \ldots, X_n\}, N)$ where t is a term and A, B, $X_1, ..., X_n$ and N are SLTs. A is the SLT positioned spatially *above* or *superscript to* t, B is the SLT positioned spatially *below* or *subscript to* t, $X_1, ..., X_n$ are the SLTs that are *arguments* to t, and N is the SLT positioned spatially to the right of t (the *next* node). We assume that a LaTeX term cannot have another term both above and superscript to it, or both below and subscript to it. No node's term may equal $\emptyset$ (the empty set) but any node itself may equal $\emptyset$. We use the notation (t) as shorthand for SLTs that only contain a term, and (t, N) for SLTs that only contain a term and a *next* branch: $(t) = (t, \emptyset, \emptyset, \emptyset, \emptyset), (t, N) = (t, \emptyset, \emptyset, \emptyset, N)$. For example, $x^2 + 5 * \sqrt{y_1} = (x, (2), \emptyset, \emptyset, (+, (5, (*, (\backslash sqrt, \emptyset, \emptyset, (y, \emptyset, (1), \emptyset, \emptyset), \emptyset)))))$.

The *sub-expressions* of an expression are all of the SLTs contained within Above, Below and Argument branches, as well as those within parentheses, brackets and braces. Sub-expressions themselves are also expressions and can contain sub-expressions of their own. For example, the sub-expressions of $x_{n-1}^{x*(y+1)}$ are $n - 1$, $x * (y + 1)$ and $y + 1$.

A *substitution* $\sigma = \{S_1 \to T_1, ..., S_n \to T_n\}, \forall i \; S_i \in \mathbf{V}$ represents a replacement of each SLT S_i with the corresponding SLT T_i. $\mathbf{V}$ is the set of *variable symbols*: single-node SLTs which will hereafter be represented by

$\boxed{\text{N}}$, where N is an integer. When a substitution σ is *applied* to an SLT X, each instance of $S_i \in \sigma$ in X is replaced with $T_i \ \forall i$. For example, if $\sigma = \{\boxed{1} \to x^2 = (x, (2), \emptyset, \emptyset, \emptyset),\ \boxed{2} \to 5 = (5)\}$ and $X = \boxed{1} + \boxed{2} = (\boxed{1}, (+, (\boxed{2})))$, $apply(\sigma, X) = X\sigma = (x, (2), \emptyset, \emptyset, (+, (5))) = x^2 + 5$.

A *substitution tree* represents a set of substitutions where each node is a substitution and each branch is a set of idempotent substitutions. (A substitution $\sigma = \{S_1 \to T_1, ..., S_n \to T_n\}$ is idempotent if $\forall i\ S_i$ does not appear in T_i). Each branch in the tree represents a binding chain for variables, and, for any node, the composition of all substitutions in the branch from the root to that node produce an expression that is an instance of the root node's substitution. Thus any node itself can be said to contain an expression which is found by composing the substitutions from the root to that node. Any mathematical expression may be inserted into a substitution tree where it becomes represented by a new node if it's not already in the tree. This insertion may also change existing nodes in order to create the most generalized set of substitutions for all expressions contained within the tree. The order in which the expressions are inserted impacts the layout of the substitution tree. A substitution tree node is described by a tuple (τ, Σ) where τ is the substitution represented by the node and Σ is the set of child substitution tree nodes.

3. EXPRESSION INDEXING

Indexing mathematical expressions using substitution trees involves inserting all of the expressions, one at a time, into a single substitution tree. This constructs a substitution tree (the *index*) that represents each expression.

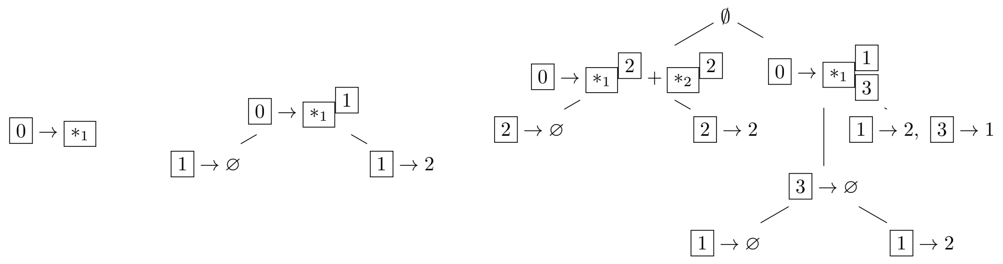
Before a new expression is inserted into a substitution tree, all variables in the expression (*mathematical variables*) must be *normalized* by renaming them as *indicator variables*, denoted by $\boxed{*_i}$. An indicator variable is a type of substitution variable that can only be substituted for a mathematical variable. Any indicator variable that is not already bound can be substituted for any mathematical variable; so $\boxed{*_i} = a = \ldots = z$. Variables are normalized in the order they appear in the input expression from left to right: for example, $x + y$ and $y + x$ are both normalized to $\boxed{*_1} + \boxed{*_2}$. This normalization allows a single node to represent multiple expressions, and this overlap helps both indexing (since fewer nodes makes the tree smaller and insertion faster) and retrieval (since expressions with such similarity should be relevant search results: x can equal y in the right context).

Inserting an expression into a substitution tree relies on the property that every child node is an instance of its parent and every parent node is a generalization of all its children. The insertion algorithm takes a normalized expression e and first looks for a *match* among the index root and its children. Two expressions match if one is an instance of another. If any of the root's children are a match, the algorithm is called recursively using that child as the new root; if the root alone is a match, a new node representing e is created and added to the root's children. If none of the nodes are a match, then the algorithm finds the *most specific common generalization* (MSCG) of e and the root, where the MSCG is a substitution containing the most specific (the closest match) of all of the possible generalizations of the two expressions. This generalization replaces the root node and takes two children of its own: an instance of the root (and its original children) and an instance of e (a leaf node that represents e). Further implementation details can be found in Schellenberg's thesis.[8]

3.1 The Indexing Algorithm

The most significant addition we made to the indexing algorithm[1] was to introduce an insertion bias. A bias is needed because otherwise most of the expressions become children of the index root. This is probably due to our transition from predicates to expressions and the sheer size of our index. This behavior is not desired because it does not take full advantage of the substitution tree's structure and would make retrieval inefficient (often forcing the algorithm to run an exhaustive search of the index). The bias we have chosen is the size of the *baseline*, which is the set of SLT nodes starting from the root of an expression and continuing along the Next branch of each node. The baseline size is not affected by the number of non-empty Above, Below or Argument branches that extend from the baseline nodes, nor their depth: thus x and $x^{x^{y}_{z}}$ both have a baseline size of 1, and $x + y$ and $\cos x + \frac{1}{2}$ both have a baseline size of 3. An SLT node and its corresponding term are said to lie along the baseline if it is part of this set. This bias causes each of the index root's sub-trees to represent expressions of a different baseline size. This specific bias is desirable because it helps group similar expressions together (based

Figure 2. *Left:* The expression x is inserted into an empty substitution tree. The tree now contains one node with a substitution from the root substitution variable $\boxed{0}$ to the normalized expression. This substitution will not only match x but also any single mathematical variable. *Middle:* Now x^2 is inserted into the same substitution tree. Since no match could be found, it creates a generalization – a new node replacing the old root and its children. The two new children nodes represent more specifics instance of this new, generalized substitution. *Right:* The same substitution tree after the expressions $x + y$, $x^2 + y^2$ and x_1^2 are added. A null substitution $\emptyset$ is sometimes produced if a generalized substitution is empty. A substitution to $\varnothing$ represents a replacement of that substitution variable with nothing.


$\boxed{0} \to \boxed{*_1}$
$\boxed{0} \to \boxed{*_1}^{\boxed{1}}$
$\boxed{1} \to \varnothing$
$\boxed{1} \to 2$
$\emptyset$
$\boxed{0} \to \boxed{*_1}^{\boxed{2}} + \boxed{*_2}^{\boxed{2}}$
$\boxed{0} \to \boxed{*_1}^{\boxed{1}}_{\boxed{3}}$
$\boxed{2} \to \varnothing$
$\boxed{2} \to 2$
$\boxed{1} \to 2,\ \boxed{3} \to 1$
$\boxed{3} \to \varnothing$
$\boxed{1} \to \varnothing$
$\boxed{1} \to 2$

on their size) which is useful for both indexing and retrieval (since large expressions are less relevant to small search queries, and visa-versa).

While a single substitution variable can't match multiple terms along the baseline due to this bias, it can still match multiple terms that are above, below, or argument to the terms along the baseline (in other words, non-baseline sub-expressions). For example, the expression $\cos(\boxed{1})$ can match $\cos(x + 1)$ using the matcher $\boxed{1} \to x + 1$, but the expression $\boxed{1}$ cannot match $x + 1$ because it lies along the baseline. Additionally, a substitution variable that does not lie along the baseline can match the empty substitution $\varnothing$, allowing us to match expressions like x^2 with x because $x = x^{\varnothing}$. This behavior is an inherent part of the algorithm and is only superseded for nodes along the Next branch due to the baseline bias we have implemented.

This algorithm has a few shortcomings. First, it cannot consider semantic similarities when inserting an expression, so, for example, 1+2 will not be matched to 2+1. Also, the comparison of expressions during matching does not find the largest common subsequence of the two expressions, but instead always starts comparing from the root of the SLT; therefore $2 + 1$ and $3 + 2 + 1$ will not be matched either. However, these expressions will likely still be returned as relevant results during the retrieval process as described below.

4. EXPRESSION RETRIEVAL

Retrieving relevant mathematical expressions from substitution trees benefits from the properties of the tree and the behavior of insertion. Finding exact matches to a *search query* (query-by-expression) is straightforward because, if the expression exists in the index, it can be found by simply following the correct branch (the branch that contains nodes that match the search query). Expressions similar to the search query can be found while following the matching branch due to the nature of how they were inserted. Since a new expression is inserted into a branch that shares a common match or MSCG, that expression will share the most similarities with its parent and siblings. These similarities are based on the expressions' shared symbols and symbol layout (the arrangement of those symbols). Thus the parents and siblings of a node that matches the search query will also be relevant search results.

The search algorithm works by seeing if the search query is a match to the index root. If so, the algorithm is called recursively on each of the root's children; if not, the search fails. If the root is both a match and a leaf, the expression it represents is added to the list of results. Further implementation details can be found in Schellenberg's thesis.[8]

4.1 The Retrieval Algorithm

Our innovations for Graf's retrieval algorithm include searching not only on the query (which will only return exact matches), but also on each of the query's sub-expressions,[4] and several variations of the query and its sub-expressions that are generated by the retrieval algorithm. Thus our system can find expressions that are both identical to and relevant to (unifiable with) the query.

We make multiple variations of the search query by *extending* it in different ways. An expression is extended by adding one or more unused substitution variables to each side of the expression; the extension amount is set by the user and every possible variation is created and used in the retrieval. For example, for the expression $x+1$ and extension amount 2, the variations include $\boxed{1}\,x+1$, $x+1\,\boxed{1}$, $\boxed{1}\,\boxed{2}\,x+1$, $\boxed{1}\,x+1\,\boxed{2}$, and $x+1\,\boxed{1}\,\boxed{2}$. This is important in order to find similar expressions that are longer than the search query, such as $x+1+2$, because the baseline size bias implemented in the indexing algorithm would otherwise cause these expressions to be ignored by the retrieval algorithm. For example, the expression $\boxed{1}\,x+1$ matches $2x+1$ which is similar (and thus relevant) to the original search query $x+1$.

While our first implementation of the retrieval algorithm worked fairly well, it overlooked a few very relevant expressions during our preliminary test runs. We changed the algorithm so that, in addition to searching the index for the query, sub-expressions of the query, and extended variations of the query and its sub-expressions, it also searches for variations of the query's *completely generalized baseline.* We create the completely generalized baseline by replacing each term on the baseline of the original search query with a substitution variable. For example, x^2+1 becomes $\boxed{1}\,\boxed{2}\,\boxed{3}$. We ignore the query's Above, Below and Argument branches during this creation because a substitution variable that lies along the baseline can match terms with or without these branches ($\boxed{1}$ can match any expression with a baseline size of 1, including x, x^2 or $x^{x^y_z}$). We also create completely generalized baselines for the extended variations of the query (so, continuing our example, we would also search for $\boxed{1}\,\boxed{2}\,\boxed{3}\,\boxed{4}$ and $\boxed{1}\,\boxed{2}\,\boxed{3}\,\boxed{4}\,\boxed{5}$, assuming the default extension amount of 2).

Notice that the use of the completely generalized baseline causes many of the previous searches to become unnecessary, because, for example, the results returned by a search for x^2+1 will always be a subset of the results returned by a search for $\boxed{1}\,\boxed{2}\,\boxed{3}$. In practice, this means that all expressions in the index with the same baseline size as the query (as well as slightly larger sizes, due to the extended queries) are added to the search result vector. While our ranking algorithm ensures that irrelevant results do not appear among the top search results, this addition essentially produces a semi-exhaustive search of the index and does not fully utilize the generality of Graf's search function (and which we could not fix due to time constraints).

The search results are ranked by comparing each individual result to the original search query, even if the result was found using a sub-expression or generalization. Comparisons for the ranking algorithm is based on a hybrid of the *bipartite expression representation*[6] and the well-known *bag-of-words* model. Comparing expressions using the bipartite representation makes a list of neighbor relationships for each of the two expressions. Each element in these lists is a 5-tuple (s, n, r, p, b), where s is the symbol, n is a symbol neighboring s, r is the relationship between s and n (above, below, argument or next), p is the position of s along the baseline, and b is a number representing the baseline that gets changed for sub-expressions. For example, $x^{x+1} = \{(x, x, above, 1, 1), (x, +, above, 1, 1), (x, 1, above, 1, 1), (x, +, next, 1, 2), (x, 1, next, 1, 2), (+, 1, next, 2, 2)\}$. The bag-of-words approach also makes a list for each expression, but the elements of this list are merely each of the symbols that appear in the corresponding expression. For example, $x^{x+1} = \{x, x, +, 1\}$.

Both comparison functions calculate a rank between 0 and 1 by finding the set similarity[7] on the number of matches and partial matches between the two lists. Partial matches are found as a percentage matching of tuple elements. Elements themselves can either match fully if they are equal or partially if their terms share a common type (variable, constant, operator, function), so y is a closer match to x than 1; for example, if $A = (x, +, next, 1, 1)$, $B = (y, +, next, 1, 1)$, $C = (1, +, next, 1, 1)$, then $rank(A, A) = 1.0$, $rank(A, B) = 0.85$, and $rank(A, C) = 0.8$. Partial matches are chosen by a greedy matching algorithm. Using the set similarity forces both the number of successful matches and the length of each expression to factor in to the rank. It also causes identical matches to always be ranked at 100%.

5. EXPERIMENTS AND RESULTS

We used the *precision-at-k* performance metric (which measures the relevance of the top k results retrieved during a search[3]) to test the top $k = 20$ results of 10 test search queries. Our search queries and document database were the same that Zanibbi and Yuan used in their experiment.[5] We evaluated our results through an online

Table 1. Precision-at-k ($k = 20$) for 10 search queries (collection: 24,479 expressions from 50 LaTeX documents). Ten participants rated the results as “not similar at all” (0), “somewhat similar” (0.5) or “very similar/identical” (1.0). The mean ratings (including standard deviation, abbreviated as SD) for the top 5 results and top 20 results for Zanibbi and Yuan’s Lucene keyword-based system[5] and our substitution tree system (“ST”) are shown below as percentages.

		Top 5 Lucene		Top 5 ST		Top 20 Lucene		Top 20 ST	
Query	Expression	Mean	SD	Mean	SD	Mean	SD	Mean	SD
1	$d,$	80.0	0.0	40.0	33.5	78.3	11.3	29.8	16.7
2	$L_1 \times L_2 \times L_3$	43.0	37.4	43.0	39.0	18.3	24.7	17.3	23.8
3	$\{v \in W_0^{1,p}(D) :\ v \geq \psi \text{ a.e. in } D\}$	49.0	36.7	37.0	43.4	25.8	27.2	15.8	23.8
4	e_{n+1}	32.0	32.5	95.0	5.0	38.0	38.0	65.0	25.6
5	$\mathcal{U}'$	50.0	34.8	47.0	30.3	22.5	25.8	18.8	23.4
6	$\prod_{y \in \Sigma} k(y) \rightarrow \prod_{y \in \Sigma'} k(y)$	40.0	35.5	34.0	41.7	13.3	23.9	9.0	24.3
7	$\mathbf{x}^{\mathbf{u}_1}$	61.0	21.6	38.0	48.0	26.3	29.4	12.5	27.5
8	$D^n = CS^{n-1}$	38.8	50.0	35.8	37.8	11.5	28.2	13.0	22.2
9	$\Omega_{\tau a}$	69.0	34.9	78.0	21.7	31.0	32.3	29.8	37.0
10	$\Omega = \frac{h^2}{m}(k_1^2 + k_2^2) + v_0 + v_{2k_2},$	49.0	38.3	37.0	47.1	14.3	27.4	10.5	26.9
	Mean (μ)	51.1	32.2	48.4	34.8	27.9	26.8	22.2	25.1
	Standard Deviation (σ)	14.9	13.2	20.8	13.2	19.6	6.8	16.7	5.1

survey that was completed by 10 college upperclassmen in the Computer Science, Math and Engineering fields (students with an advanced knowledge of mathematics). The survey contained 20 questions randomly ordered for each participant to prevent effects arising from presenting retrieval results in a fixed order. Each of the 10 search queries appeared twice: once with the top 20 results using Zanibbi and Yuan’s Lucene keyword-based system, and again with the top 20 results using our substitution tree system. All expressions were shown as pictures created using `latex2html`. Each question asked the participant to individually judge the 20 search results on their similarity to the query “in terms of both the similarity in symbols between the query and candidate expressions and in their spatial arrangement.” Each search result was rated as either “not similar at all,” “somewhat similar” or “very similar/identical.” This rating was converted to the numerical scale (0, 0.5, 1.0) and averaged for each result to calculate the mean rating. The mean rating of the top 5 and top 20 results for each query was averaged to produce the final results that appear in Table 1.

We indexed 24,479 expressions from 50 LaTeX documents into a 12 GB file in about 6 minutes using a server with 2 Intel Xeon X5670 2.93 GHz processors with 24 cores and a total of 95 GB of RAM. Our system contained fewer expressions than Zanibbi and Yuan’s because it ignores expressions containing only one symbol (which appear far too frequently to be useful) or over 100 symbols (which take up far too much space). Retrieval took between 2 seconds for query 1 to 2.5 minutes for query 10 (except for query 3, which took 12 minutes), largely dependent on the size of the search query and its sub-expressions, and returned between 4992 (for query 2) to 13266 (for query 6) search results. Queries with more sub-expressions seemed to return more results due to the extra searches made using those sub-expressions.

The substitution tree system performs comparably to the Lucene system in many cases, but can also perform much better or much worse. However, the top 1 result was rated “identical” for all 10 search queries using the substitution tree system, but for only 7 using the Lucene system (queries 1, 4, and 7 did not list the identical match first, but did list it in their top 20). Table 2 compares the top 20 results from each system for query 4.

Table 2. Top 20 Results for Query 4

Result	Lucene	Sub. Tree	Result	Lucene	Sub. Tree
1	$e_{n+1}, e_{n+2}\}$	e_{n+1}	11	$= 2v_{43} + v_{46} - \alpha_{13}\alpha_7 v_{58},$	$\frac{1}{\sqrt{\alpha}}e_{n+1}$
2	$e_{n+1},\ e_{n+2}\},$	e_{n+1}	12	$e_1, e_2, \cdots, e_{n+1}$	$\mathscr{O}_{n+1}$
3	$e_{n+2} + \alpha e_1 + e_2$	$e_{n+1},$	13	$V^{\mathfrak{f}}_{43} = \langle \alpha_{13}v_{30} + v_{31}, v_{32}, v_{33} + \alpha_7\alpha_{13}v_{42}, v_{34}, \ldots, v_{41}, v_{58}\rangle,$	$e_{r+1},$
4	$\nu_1 + \nu_2 + \nu_3 = 2$	$e_{n+1}]$	14	$e_{r+1}, \cdots, e_{n+2}$	$e_{n+2},$
5	$\nu_1 + \nu_2 + \nu_3 = d$	$e_{n+1}\}$	15	$b^1_{ij}e_1 + b^2_{ij}e_2 = [e_1, e_2, \cdots, \hat{e_i}, \cdots, \hat{e_j}, \cdots, e_{n+1}, e_{n+2}]$	$e_{n+2}\}$
6	$e_{n+1},$	e_{n+2}	16	$\mathfrak{n} = \mathfrak{g}_{\alpha_1} \oplus \mathfrak{g}_{\alpha_1+\alpha_2} \oplus \mathfrak{g}_{2\alpha_1+\alpha_2} \oplus \mathfrak{g}_{3\alpha_1+\alpha_2} \oplus \mathfrak{g}_{3\alpha_1+2\alpha_2} \cong \mathbb{C}^5,$	$e_{n+2}\}$
7	e_{n+1}	e_{n+2}	17	$\mathfrak{n} = \mathfrak{g}_{\alpha_1} \oplus \mathfrak{g}_{\alpha_1+\alpha_2} \oplus \mathfrak{g}_{2\alpha_1+\alpha_2} \oplus \mathfrak{g}_{3\alpha_1+\alpha_2} \oplus \mathfrak{g}_{3\alpha_1+2\alpha_2} \cong \mathbb{R}^5,$	$e_{n+2}]$
8	e_{n+1}	$\underline{f_{n+1}}$	18	$f_i = x_1^{\nu_1+1}f_{i,1} + x_2^{\nu_2+1}f_{i,2} + x_3^{\nu_3}f_{i,3} \quad (1 \leq i \leq 3),$	$e_{n+2},$
9	$e_{n+1},$	X^{n+1}	19	$(1)'' \begin{cases} [e_2, \cdots, e_{n+1}] = e_1, \\ [e_1, e_3, \cdots, e_{n+1}] = e_2, \\ [e_1, e_2, \hat{e}_3, \cdots, e_{n+1}] = e_3, \\ [e_1, e_4, \cdots, e_{n+2}] = b^1_{2,3}e_1, \\ [e_2, e_4, \cdots, e_{n+2}] = b^1_{2,3}e_2, \\ [e_3, e_4, \cdots, e_{n+2}] = b^1_{2,3}e_3. \end{cases}$	$e_{n+1},$
10	$\underline{f_{n+1}}$	$\frac{1}{a}e_{n+1}$	20	$= 2v_{23} + v_{25} - \alpha_1 v_{31} + \alpha_1\alpha_{11}v_{42} + (\alpha_1\alpha_{12} + \alpha_{11}\alpha_2 - \alpha_{13}\alpha_3)v_{58},$	$F^{n+1}.$

6. DISCUSSION

We can observe the main strength of our substitution tree system through query 4 where it outperforms its Lucene counterpart both in the mean ratings obtained from our experiment and the quality of its top 20 search results (as shown in Table 2). Our system finds many relevant results through sub-expression matching which is apparent in the consistency of results containing $n + 1$ and $n + 2$ (sometimes even as a superscript rather than a subscript). It also ranks results identical to query 4 before non-identical results, unlike the Lucene system which ranks six non-identical results higher than the identical results. This may exemplify the detriment of the TF-IDF ranking method: since TF-IDF is heavily influenced by documents in the index, it can rank relevant (and identical) results lower simply because they occur more often. While this might be a good idea for text information retrieval, it does not seem as effective for mathematical expressions, especially since a document referencing the same expression multiple times often increases its importance.

The mean top 1 result rating over all ten queries is 100% for our system and 89% for the Lucene system (the top result for queries 1, 4 and 7 was not identical). The mean ratings for both the top 5 results and top 20 results of queries 2, 5, 6, and 8 are comparable between the two systems. The Lucene system produced much better results for queries 3 and 10, suggesting that our system has difficulty with larger expressions. One problem with our experiment is that our system doesn't add single symbol expressions to its index while the Lucene system does. This is why the Lucene system does so much better in our test on query 1: 17 of its top 20 results are the expression d which most of our participants deemed "very similar/identical" to the query (d,). Since our system did not index those expressions, the results were skewed in favor of the Lucene system.

LaTeX formatting has a great effect on our system as shown in the top 20 results of queries 5, 7 and 9 and their comparatively low mean ratings. Query 5 begins with `\mathcal`, query 7 begins with `\mathbf`, and query 9 begins with `\,`. Our system emphasizes these terms (perhaps because they appear at the start of the queries) during retrieval. While our system is successful in this aspect – all of the top 20 results for the three queries begin with `\mathcal`, `\mathbf`, and `\,` respectfully – it affected the experiment because the participants were looking for shared variables rather than shared formatting and therefore rated these queries' results lower. This could have been a flaw in our experiment since we only showed our participants images of the expressions and told them to rate results based on their similarity to the query in symbols and symbol layout, not formatting. It might be prudent for our system to ignore such terms when indexing expressions because formatting has such a significant effect on our system's retrieval and ranking algorithms while having little to no effect on the visual and mathematical similarity of two expressions.

Our addition of a bias on the baseline size of an expression during its insertion into the index also had a major and noticeable impact on our system's search results. Most of the top 20 results produced by our system are visually similar in size to the query, while the results produced by the Lucene system vary dramatically. Since the searches for our system used the default extension amount of 2, their results were restricted to a baseline size equal to the query or up to two greater than the query unless they were matched on a search of one of the query's sub-expressions. This hurt our results because they seem to miss some relevant results that the Lucene system finds. While we chose this bias in part because the disparity in size between a search query and search result has a significant impact on that result's relevancy (results often become less relevant as they become larger because of all the superfluous symbols), we see from our experiment that this is not always the case, especially when the query itself (or an expression very similar to the query) is contained within a larger expression.

7. CONCLUSION

Our experiment demonstrates that our novel substitution tree MIR system is comparable to Zanibbi and Yuan's keyword-based vector-space Lucene MIR system. As exemplified in our system's perfect performance in the top 1 search results, our system's ranking algorithm, which is based only on mathematical similarity, seems to work much better than the Lucene system's ranking algorithm, which is based in part on an expression's frequency in the index. Our system's retrieval algorithm successfully finds results using the sub-expressions of search queries, providing further strength to the quality of our search results. Unfortunately, our system is negatively affected by two major factors: the presence of LaTeX formatting in our search queries and the expressions in our index, and the impact of the insertion bias on the results that our retrieval algorithm is able to find. However, these are problems that could be addressed and fixed.

Overall, we believe that we have provided a strong foundation for future research in substitution tree indexing and retrieval for mathematical expressions, and that future implementations which build and learn from our work will be extremely successful. Suggestions for future work include: ignoring LaTeX formatting during indexing and retrieval; designing a different insertion bias; extending our system to index websites or non-LaTeX expressions; adding sub-expressions to the index; and fixing the retrieval algorithm by removing the completely generalized baseline and creating alternate variations of search queries (ideas for which can be found in Schellenberg's thesis[8]).

REFERENCES

1. P. Graf, "Substitution tree indexing," in *RTA '95: Proceedings of the 6th International Conference on Rewriting Techniques and Applications*, pp. 117–131, Springer-Verlag, (London, UK), 1995.
2. B. R. Miller and A. Youssef, "Technical aspects of the digital library of mathematical functions," in *Annals of Mathematics and Artificial Intelligence*, pp. 121–136, May 2003.
3. R. Zanibbi and D. Blostein, "Recognition and retrieval of mathematics," *Int'l. Journal of Document Analysis and Recognition* , 2011 (to appear).
4. M. Kohlhase and I. Sucan, "A search engine for mathematical formulae," in *Proc. Artificial Intelligence and Symbolic Computation*, J. C. Testuo Ida and D. Wang, eds., *number 4120 in LNAI*, pp. 241–253, Springer Verlag, 2006.
5. R. Zanibbi and B. Yuan, "Keyword and image-based retrieval for mathematical expressions," in *Proc. Document Recognition and Retrieval XVIII*, *Proc. SPIE* **7874**, pp. OI1–OI9, (San Francisco, CA), Jan. 2011.
6. R. Zanibbi, A. Pillay, H. Mouchere, C. Viard-Gaudin, and D. Blostein, "Stroke-based performance metrics for handwritten mathematical expressions," in *Int'l Conf. Document Analysis and Recognition*, pp. 334–338, (Beijing), 2011.
7. S. Kamali and F. Tompa, "Improving mathematics retrieval," in *Proc. DML 2009: Towards a Digital Mathematics Library*, pp. 37–48, (Grand Bend, Canada), July 2009.
8. T. Schellenberg, "Layout-based substitution tree indexing and retrieval for mathematical expressions," Master's thesis (Computer Science), Rochester Institute of Technology, Rochester, NY.